Importance of reguliraztion:

Preventing overfitting: Regularization helps to limit the model's capacity to memorize noise in the training data, leading to better generalization on unseen data.

Dealing with high-dimensional data: In datasets with a large number of features, regularization can help select the most important features and reduce the impact of irrelevant or redundant features.

Handling collinearity: When features are highly correlated, regularization techniques can mitigate the multicollinearity problem and provide more stable and interpretable coefficient estimates.

Improving model interpretability: Regularization can drive some coefficients to zero, effectively performing feature selection and simplifying the model, making it easier to interpret.

By incorporating regularization techniques into the model training process, we can achieve a balance between fitting the training data well and avoiding overfitting, leading to more robust and generalizable models.

Type of regulirization:

1.Lasso regulirization:convert high coefficient to 0 , this way eliminate unrelated attribute L1_reg/Lasso R = loss+alpha(|w|)where alpha(|w|)is penalty, alpha is parameter and w is the vector coefficient of model,regulirization parameter control the strength of regulirization

Ridge regression/L2 regulirization: convert high coefficient to low coefficient of the model L1_reg/Lasso R = loss+alpha(|w|)2where alpha(|w|)is penalty.

Elasticnet regression: Elasticnet.R=loss+alpha1(|w|)+alpha2(|w|)2

w=w1+w2+w3......+wn Noted, After regulirization we trained the mode. In different model algo we can use Regulirization as parameter l1 and l2. for example NN, Logistic rregression SVM in here, we will use for genirilizing Lenear model

In [1]:
```
#Import numerical libraries
import pandas as pd
import numpy as np

#Import graphical plotting libraries
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

In [6]:
```
df = pd.read_csv('/Users/myyntiimac/Desktop/car-mpg.csv')
df.head()
```

Out[6]:

	mpg	cyl	disp	hp	wt	acc	yr	origin	car_type	car_name
0	18.0	8	307.0	130	3504	12.0	70	1	0	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	0	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	0	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	0	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	0	ford torino

In [7]:

```
df.shape
```

Out[7]:

```
(398, 10)
```

Insight:This data contain 398 rows and 10 columns, And by undestanding 10 columns we can tell that miles per gallon(mpg)is dependent and others nine variables are independent where we assumes car_name has no effects in dependent variable mpg, so we can delete it by drop method

Strating Basic EDA

In [8]:

```
df = df.drop(['car_name'], axis = 1)
df.tail()
```

Out[8]:

	mpg	cyl	disp	hp	wt	acc	yr	origin	car_type
393	27.0	4	140.0	86	2790	15.6	82	1	1
394	44.0	4	97.0	52	2130	24.6	82	2	1
395	32.0	4	135.0	84	2295	11.6	82	1	1
396	28.0	4	120.0	79	2625	18.6	82	1	1
397	31.0	4	119.0	82	2720	19.4	82	1	1

In [9]:

```
#Now check the null values in our df
df.isnull().any()
```

Out[9]:

```
mpg         False
cyl         False
disp        False
hp          False
wt          False
acc         False
yr          False
origin      False
car_type    False
dtype: bool
```

In [11]:

```
#check any column contain ? mark
df.isin(['?']).any()
```

Out[11]:
```
mpg         False
cyl         False
disp        False
hp           True
wt          False
acc         False
yr          False
origin      False
car_type    False
dtype: bool
```

In [12]:
```
#we find hp column contain ? mark
#now we replace the question mark with nan values then replace NAN velues wi
df= df.replace('?', np.nan)
```

In [13]:
```
#check again
df.isin(['?']).any()
```

Out[13]:
```
mpg         False
cyl         False
disp        False
hp          False
wt          False
acc         False
yr          False
origin      False
car_type    False
dtype: bool
```

In [14]:
```
#NO ? mark now , but there is NAN velues that need to fill
df['hp'].fillna(df['hp'].median(), inplace=True)
```

In [15]:
```
#check again nan value
df.isnull().any()
```

Out[15]:
```
mpg         False
cyl         False
disp        False
hp          False
wt          False
acc         False
yr          False
origin      False
car_type    False
dtype: bool
```

In [16]:
```
#now no null value and ? mark in df
#But there is anouther column name "origin" which is catagorical column but
#this column we converted to individual column by get_dummies()
#For this first we create dictionary
df['origin'] = df['origin'].replace({1: 'america', 2: 'europe', 3: 'asia'})
df.head()
```

Out[16]:

	mpg	cyl	disp	hp	wt	acc	yr	origin	car_type
0	18.0	8	307.0	130	3504	12.0	70	america	0
1	15.0	8	350.0	165	3693	11.5	70	america	0
2	18.0	8	318.0	150	3436	11.0	70	america	0
3	16.0	8	304.0	150	3433	12.0	70	america	0
4	17.0	8	302.0	140	3449	10.5	70	america	0

In [17]:
```
# now converted the origin column into 3 column which represent each contine
df = pd.get_dummies(df,columns = ['origin'])
```

In [18]:
```
df.head()
```

Out[18]:

	mpg	cyl	disp	hp	wt	acc	yr	car_type	origin_america	origin_asia	origin_europe
0	18.0	8	307.0	130	3504	12.0	70	0	1	0	0
1	15.0	8	350.0	165	3693	11.5	70	0	1	0	0
2	18.0	8	318.0	150	3436	11.0	70	0	1	0	0
3	16.0	8	304.0	150	3433	12.0	70	0	1	0	0
4	17.0	8	302.0	140	3449	10.5	70	0	1	0	0

In [37]:
```
df.shape
```

Out[37]:
```
(398, 11)
```

feature scaling

Features often have different scales, ranges, or units of measurement. Scaling helps bring all the features to a similar scale, ensuring they have a similar impact during modeling. in our df , we can see the columns in different scale , so before trainig the model with training data we have to scalized, otherwise some featerure can show more effect on model(it will be bias)

Two scaliziation tecqniques: 1) Z_score,Std.scaler/standerization=where mean of attribute convert to 0 and std.dev=1 z = (x - μ) / σ

Where:

z is the standardized value of the data point. x is the original value of the data point. μ is the mean of the feature. σ is the standard deviation of the feature. 2)Normalization /min_max scaler:values are shifted 0 to 1 x_scaled = (x - min(x)) / (max(x) - min(x))

Where:

x is the original value of the data point. x_scaled is the scaled value of the data point. min(x) is the minimum value of the feature. max(x) is the maximum value of the feature.

First we divide the data into independent (X) and dependent data (y) then we scale it. WHY?

Because *The reason we don't scale the entire data before and then divide it into train(X) & test(y) is because once you scale the data, the type(data_s) would be numpy.ndarray. It's impossible to divide this data when it's an array.*

Hence we divide type(data) pandas.DataFrame, then proceed to scaling it.

In [38]:
```
X = df.drop(['mpg'], axis = 1) # independent variable
y = df[['mpg']]
```

In [39]:
```
X
```

Out[39]:

	cyl	disp	hp	wt	acc	yr	car_type	origin_america	origin_asia	origin_europe
0	8	307.0	130	3504	12.0	70	0	1	0	0
1	8	350.0	165	3693	11.5	70	0	1	0	0
2	8	318.0	150	3436	11.0	70	0	1	0	0
3	8	304.0	150	3433	12.0	70	0	1	0	0
4	8	302.0	140	3449	10.5	70	0	1	0	0
...	...	...	...	...	...	...	...	...	...	...
393	4	140.0	86	2790	15.6	82	1	1	0	0
394	4	97.0	52	2130	24.6	82	1	0	0	1
395	4	135.0	84	2295	11.6	82	1	1	0	0
396	4	120.0	79	2625	18.6	82	1	1	0	0
397	4	119.0	82	2720	19.4	82	1	1	0	0

398 rows × 10 columns

In [40]:
```
y
```

Out[40]:

	mpg
0	18.0
1	15.0
2	18.0
3	16.0
4	17.0
...	...
393	27.0
394	44.0
395	32.0
396	28.0
397	31.0

398 rows × 1 columns

In [46]:
```
#Now we scalize the independent and dependent variable by std.scalirization
from sklearn.preprocessing import StandardScaler

# Standardize the independent variables
scaler = StandardScaler()
X_std = scaler.fit_transform(X)
X_std = pd.DataFrame(X_std, columns = X.columns) #converting scaled data int
X_std
```

Out[46]:

	cyl	disp	hp	wt	acc	yr	car_type	origin_an
0	1.498191	1.090604	0.673118	0.630870	-1.295498	-1.627426	-1.062235	0.7
1	1.498191	1.503514	1.589958	0.854333	-1.477038	-1.627426	-1.062235	0.7
2	1.498191	1.196232	1.197027	0.550470	-1.658577	-1.627426	-1.062235	0.7
3	1.498191	1.061796	1.197027	0.546923	-1.295498	-1.627426	-1.062235	0.7
4	1.498191	1.042591	0.935072	0.565841	-1.840117	-1.627426	-1.062235	0.7
...	...	...	...	...	...	...	...	
393	-0.856321	-0.513026	-0.479482	-0.213324	0.011586	1.621983	0.941412	0.7
394	-0.856321	-0.925936	-1.370127	-0.993671	3.279296	1.621983	0.941412	-1.2
395	-0.856321	-0.561039	-0.531873	-0.798585	-1.440730	1.621983	0.941412	0.7
396	-0.856321	-0.705077	-0.662850	-0.408411	1.100822	1.621983	0.941412	0.7
397	-0.856321	-0.714680	-0.584264	-0.296088	1.391285	1.621983	0.941412	0.7

398 rows × 10 columns

In [48]:

```
scaler = StandardScaler()
y_std = scaler.fit_transform(y)
y_std = pd.DataFrame(y_std, columns = y.columns) #converting scaled data int
y_std
```

Out[48]:

	mpg
0	-0.706439
1	-1.090751
2	-0.706439
3	-0.962647
4	-0.834543
...	...
393	0.446497
394	2.624265
395	1.087017
396	0.574601
397	0.958913

398 rows × 1 columns

Insight:The values in column 0 have been transformed such that the mean is 0 and the standard deviation is 1. The first value (-0.856) is below the mean and the other value (1.498) is above the mean.

In [49]:
```
#now we split the both variable for training data and test data
# for this we import test_train_split() from skleran_model_selection
from sklearn.model_selection import train_test_split
X_train, X_test, y_train,y_test = train_test_split(X_std, y_std, test_size =
X_train.shape
```

Out[49]:
```
(298, 10)
```

In [ ]:
```
#Now build the model
#First Lenear model then  lasso and ridge and compare the performance of the
# FOr first lenear model we use enamurate() to see how model coefficient loc
#and decrease the overfitting problem and improve the model
```

In [50]:
```
from sklearn.linear_model import LinearRegression

# Create and fit the linear regression model
regression_model = LinearRegression()
regression_model.fit(X_train,y_train)

# Print the coefficients for each independent variable
#oefficients for each independent variable are printed using a for loop comb
for col_name, coef in zip(X_train.columns, regression_model.coef_[0]):
    print(f"The coefficient for {col_name} is {coef}")

# Print the intercept
#The intercept is stored in the intercept variable and printed using an f-st
intercept = regression_model.intercept_[0]
print(f"The intercept is {intercept}")
```

```
The coefficient for cyl is 0.295188274518623
The coefficient for disp is 0.34377235544447493
The coefficient for hp is -0.2031300205066636
The coefficient for wt is -0.7282388801097291
The coefficient for acc is 0.03466225460648925
The coefficient for yr is 0.3805726133418513
The coefficient for car_type is 0.36210864572959445
The coefficient for origin_america is -0.08228452280158736
The coefficient for origin_asia is 0.05237035868641345
The coefficient for origin_europe is 0.04973173878319417
The intercept is 0.021346953567712677
```

In [52]:
```
#Ridge regression
#alpha factor here is lambda (penalty term) which helps to reduce the magnit
from sklearn.linear_model import Ridge
ridge_model = Ridge(alpha = 0.3)
ridge_model.fit(X_train, y_train)

print('Ridge model coef: {}'.format(ridge_model.coef_))
```

```
Ridge model coef: [[ 0.292303    0.33174595 -0.20338688 -0.7180704   0.03282
392  0.37944512
   0.3587562  -0.08172261  0.05226426  0.04912861]]
```

Insight:we find that coefficient is changed

In [53]:
```
#Lets check in Lasso regression
from sklearn.linear_model import Lasso
Lasso_model = Lasso(alpha = 0.1)
Lasso_model.fit(X_train, y_train)

print('Lasso model coef: {}'.format(Lasso_model.coef_))
```

```
Lasso model coef: [-0.          -0.          -0.02288344 -0.52067427  0.
0.2890683
  0.11160748 -0.02891466  0.          0.          ]
```

Insight: we found some coegffient are 0, thats way lasso , eleminate or ignore the complex coefficient

In [55]:
```
# now compare the model with R square value
#The score() method returns the R-squared value,
#Simple Linear Model
print(regression_model.score(X_train, y_train))
print(regression_model.score(X_test, y_test))

print('*************************')
#Ridge
print(ridge_model.score(X_train, y_train))
print(ridge_model.score(X_test, y_test))

print('*************************')
#Lasso
print(Lasso_model.score(X_train, y_train))
print(Lasso_model.score(X_test, y_test))
```

```
0.8343520392348843
0.8575776228871523
*************************
0.8343385446094193
0.8582584230696331
*************************
0.7952115294892155
0.854782394138887
```

In [ ]:
```
Insight:Both Ridge & Lasso regularization performs very well on this data, t
```

In [ ]:
```
#Model parameter tuning
#r^2 is not a reliable metric as it always increases with addition of more a
#Instead we use adjusted r^2 which removes the statistical chance that impro
#Scikit does not provide a facility for adjusted r^2...
#so we use statsmodel, a library that gives results similar to what you obta
```

In [58]:
```
import statsmodels.api as sm
# Prepare the data
X = X_train[['cyl', 'disp', 'hp', 'wt', 'acc', 'yr', 'car_type', 'origin_ame
y = y_train['mpg']

# Add constant term to the independent variables
X = sm.add_constant(X)

# Fit the OLS model
ols_model = sm.OLS(y, X).fit()

# Get the parameter estimates
params = ols_model.params
params
```

Out[58]:
```
const              0.021347
cyl                0.295188
disp               0.343772
hp                -0.203130
wt                -0.728239
acc                0.034662
yr                 0.380573
car_type           0.362109
origin_america    -0.082285
origin_europe      0.049732
origin_asia        0.052370
dtype: float64
```

In [60]:
```
ols_model.summary()
```

Out[60]:

OLS Regression Results

Dep. Variable:	mpg	**R-squared:**	0.834
Model:	OLS	**Adj. R-squared:**	0.829
Method:	Least Squares	**F-statistic:**	161.2
Date:	Tue, 13 Jun 2023	**Prob (F-statistic):**	6.34e-107
Time:	03:54:43	**Log-Likelihood:**	-159.88
No. Observations:	298	**AIC:**	339.8
Df Residuals:	288	**BIC:**	376.7
Df Model:	9		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.0213	0.024	0.872	0.384	-0.027	0.070
cyl	0.2952	0.106	2.776	0.006	0.086	0.504
disp	0.3438	0.124	2.779	0.006	0.100	0.587
hp	-0.2031	0.076	-2.680	0.008	-0.352	-0.054
wt	-0.7282	0.086	-8.476	0.000	-0.897	-0.559
acc	0.0347	0.039	0.900	0.369	-0.041	0.110
yr	0.3806	0.028	13.454	0.000	0.325	0.436
car_type	0.3621	0.065	5.586	0.000	0.235	0.490
origin_america	-0.0823	0.020	-4.202	0.000	-0.121	-0.044
origin_europe	0.0497	0.021	2.392	0.017	0.009	0.091
origin_asia	0.0524	0.020	2.647	0.009	0.013	0.091

Omnibus:	17.430	**Durbin-Watson:**	2.142
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	23.858
Skew:	0.439	**Prob(JB):**	6.60e-06
Kurtosis:	4.072	**Cond. No.**	6.69e+15

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The smallest eigenvalue is 3.75e-29. This might indicate that there are
strong multicollinearity problems or that the design matrix is singular.

In [61]:
```
# calculate MSE from predicted Y and actual Y_test
Y_pred=regression_model.predict(X_test)
Y_pred
```

Out[61]:
```
array([[-5.45633545e-01],
       [ 6.05224257e-01],
       [-2.95500865e-01],
       [ 6.02203961e-01],
       [-9.01910406e-02],
       [-8.37764197e-01],
       [ 8.67029488e-01],
       [ 1.50701925e+00],
       [-6.28044713e-01],
       [-1.58460998e+00],
       [ 9.18187719e-01],
       [-6.37225904e-01],
       [-4.30911078e-01],
       [ 4.14159104e-01],
       [ 1.74426214e+00],
       [-3.75431930e-03],
       [-1.62418193e+00],
       [-6.25345918e-01],
       [-1.82223398e+00],
       [ 1.32148818e+00],
       [ 3.71511216e-01],
       [ 1.09488132e+00],
       [-5.40607558e-01],
       [ 2.60428937e-01],
       [ 3.72999854e-01],
       [ 9.26136877e-01],
       [ 1.23596475e+00],
       [ 1.29715456e+00],
       [-9.60468185e-01],
       [ 8.74402695e-01],
       [ 1.97066475e-01],
       [-1.68249676e+00],
       [-2.24031935e-01],
       [ 6.92675135e-01],
       [ 2.49334561e-01],
       [-1.20816136e+00],
       [ 4.58206904e-01],
       [-1.88550614e+00],
       [ 1.06084984e+00],
       [ 1.56106357e-01],
       [ 1.41286332e-01],
       [ 1.92268928e-01],
       [-1.75606207e-01],
       [ 1.33575371e+00],
       [ 1.79737791e-03],
       [-4.05584061e-01],
       [-4.93245447e-01],
       [-1.42040779e+00],
       [ 7.25390608e-01],
       [-7.98713325e-01],
       [ 1.57375909e-01],
       [ 4.12742795e-01],
       [-7.13208661e-01],
       [-1.34085256e+00],
       [ 6.72307620e-01],
       [ 2.37617863e-01],
       [-1.71012208e+00],
       [-1.21581935e+00],
       [ 9.83867271e-01],
       [ 1.63431074e+00],
       [ 1.63346651e+00],
       [ 1.64312033e+00],
       [-6.01732409e-01],
       [ 2.09207571e-01],
```

```
       [-2.86040505e-01],
       [ 1.24485074e+00],
       [ 7.20692290e-02],
       [ 1.83589350e-01],
       [ 8.41495172e-01],
       [-1.27985279e+00],
       [-2.62295832e-01],
       [-5.39652159e-02],
       [-1.11067555e+00],
       [ 4.46663210e-01],
       [-1.38343143e+00],
       [ 3.18637749e-01],
       [ 9.13874982e-01],
       [-7.67583552e-01],
       [-1.32838570e+00],
       [-1.18141950e-02],
       [-9.62087107e-02],
       [-6.01430932e-01],
       [ 1.61760033e+00],
       [ 9.78412826e-02],
       [ 1.50019976e+00],
       [-9.66352149e-01],
       [-8.58942279e-01],
       [-1.58472597e-01],
       [ 1.55463479e+00],
       [ 1.12368955e+00],
       [-3.54978399e-01],
       [-3.72939663e-01],
       [-1.83544510e-01],
       [ 5.36823685e-01],
       [ 1.01009679e+00],
       [-3.92142556e-01],
       [ 1.33078142e+00],
       [ 6.84414144e-01],
       [ 6.77705360e-02],
       [-7.29293202e-02]])
```

In [63]:
```
# Calculate Mean Squared Error (MSE)
mse = np.mean((Y_pred - y_test) ** 2)
mse
```

```
/Users/myyntiimac/anaconda3/lib/python3.10/site-packages/numpy/core/fromnumeric.py:3430: FutureWarning: In a future version, DataFrame.mean(axis=None) will return a scalar mean over the entire DataFrame. To retain the old behavior, use 'frame.mean(axis=0)' or just 'frame.mean()'
  return mean(axis=axis, dtype=dtype, out=out, **kwargs)
```

Out[63]:
```
mpg    0.128032
dtype: float64
```

In [64]:
```
# calculate Rooot Mean square error
import math
rmse = math.sqrt(mse)
rmse
```

Out[64]:
```
0.35781548331897894
```

Insight: so the difference between actual mpg and predicted mpg is .357

Chceck the quality of regression model

Residual plots are useful for evaluating the assumptions of a regression model.

They show the difference between the observed values and the predicted values (i.e., the residuals) on the y-axis,

while the x-axis represents the independent variable values. The lowess curve in the plot provides a smoothed representation of the residuals,

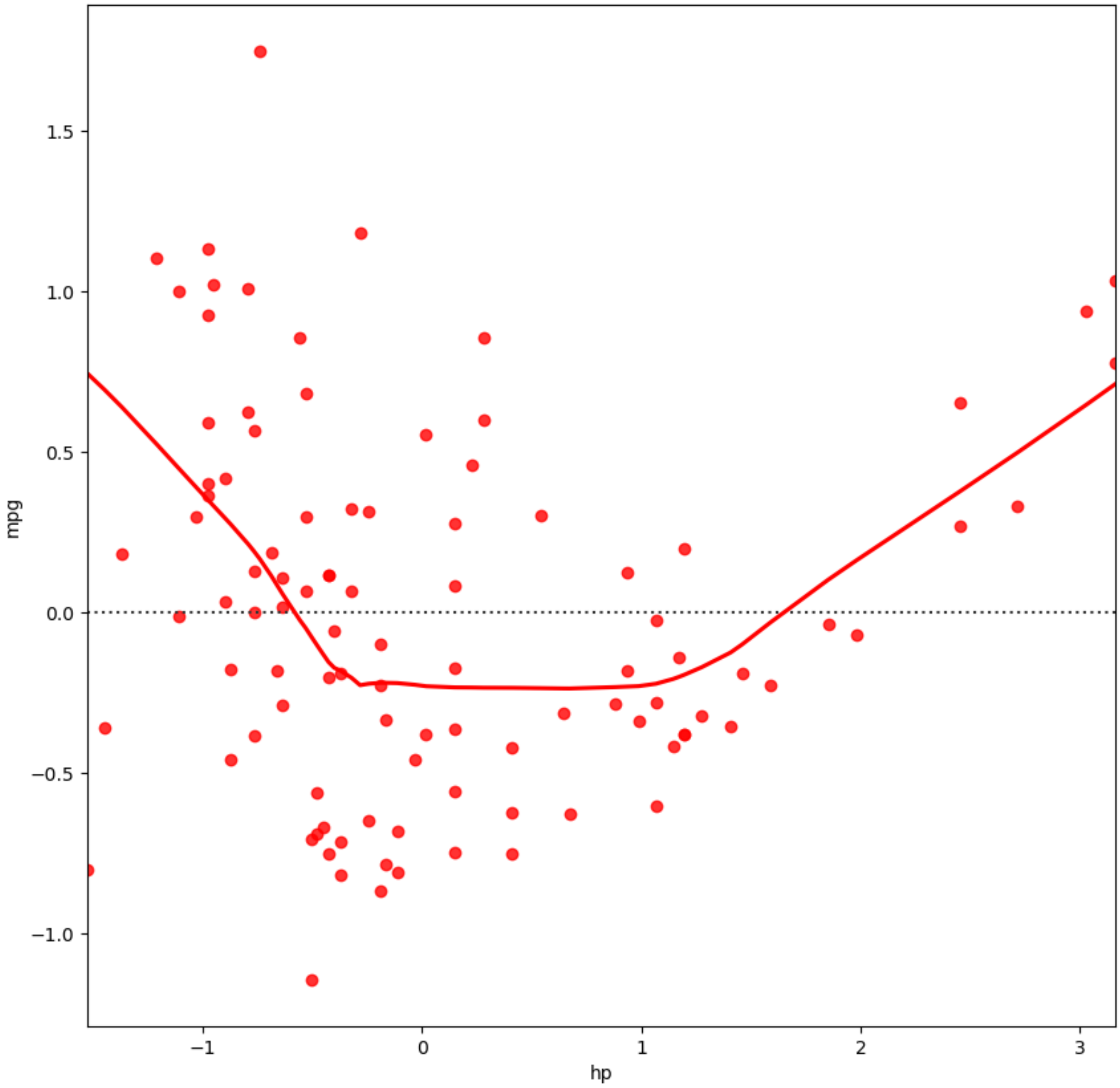
which can help identify patterns or nonlinear relationships between variables.

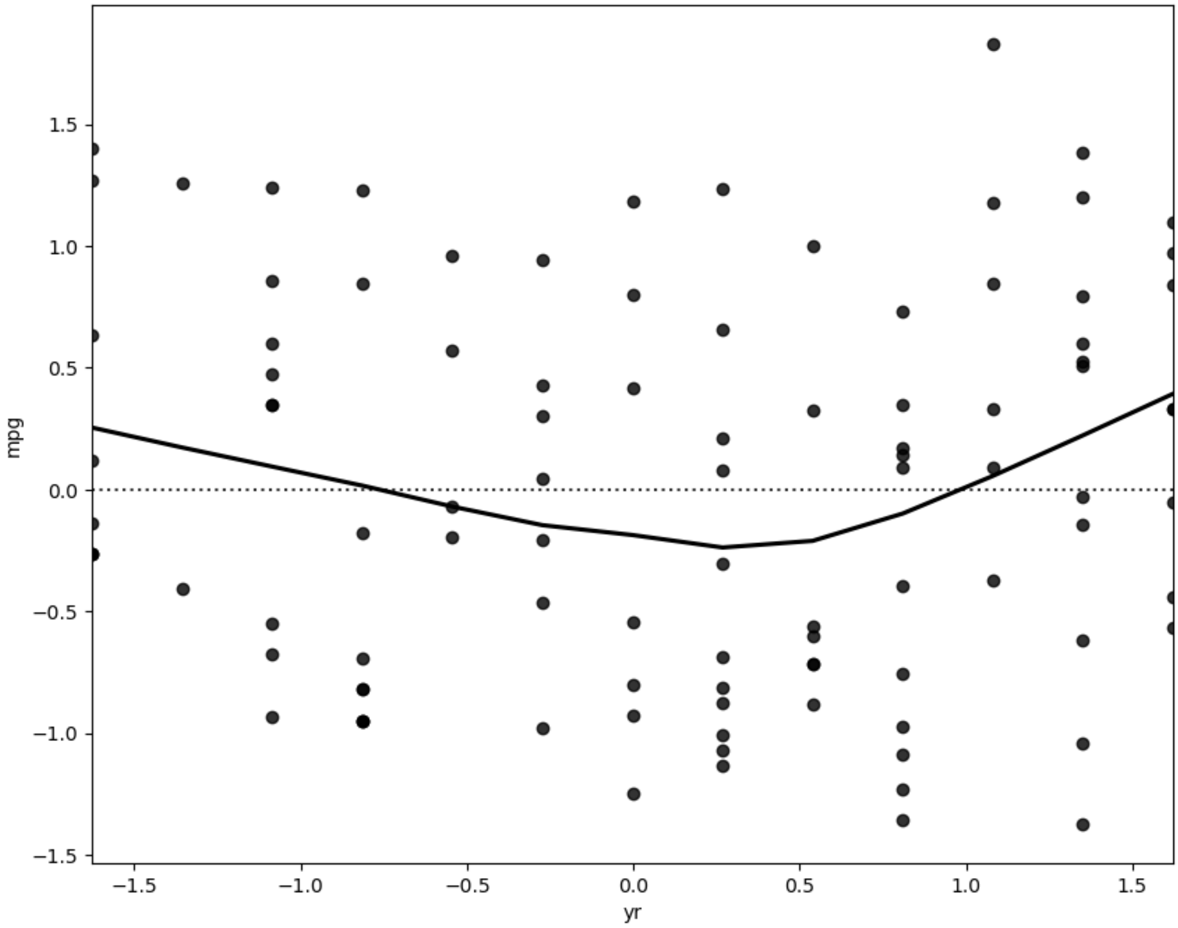
In [66]:
```
#Lets check the residuals for some of these predictor.

fig = plt.figure(figsize=(10,10))
sns.residplot(x= X_test['hp'], y= y_test['mpg'], color='red', lowess=True )


fig = plt.figure(figsize=(10,8))
sns.residplot(x= X_test['yr'], y= y_test['mpg'], color='black', lowess=True
```

Out[66]:
```
<Axes: xlabel='yr', ylabel='mpg'>
```


mpg
1.5
1.0
0.5
0.0
−0.5
−1.0
−1
0
1
2
3
hp

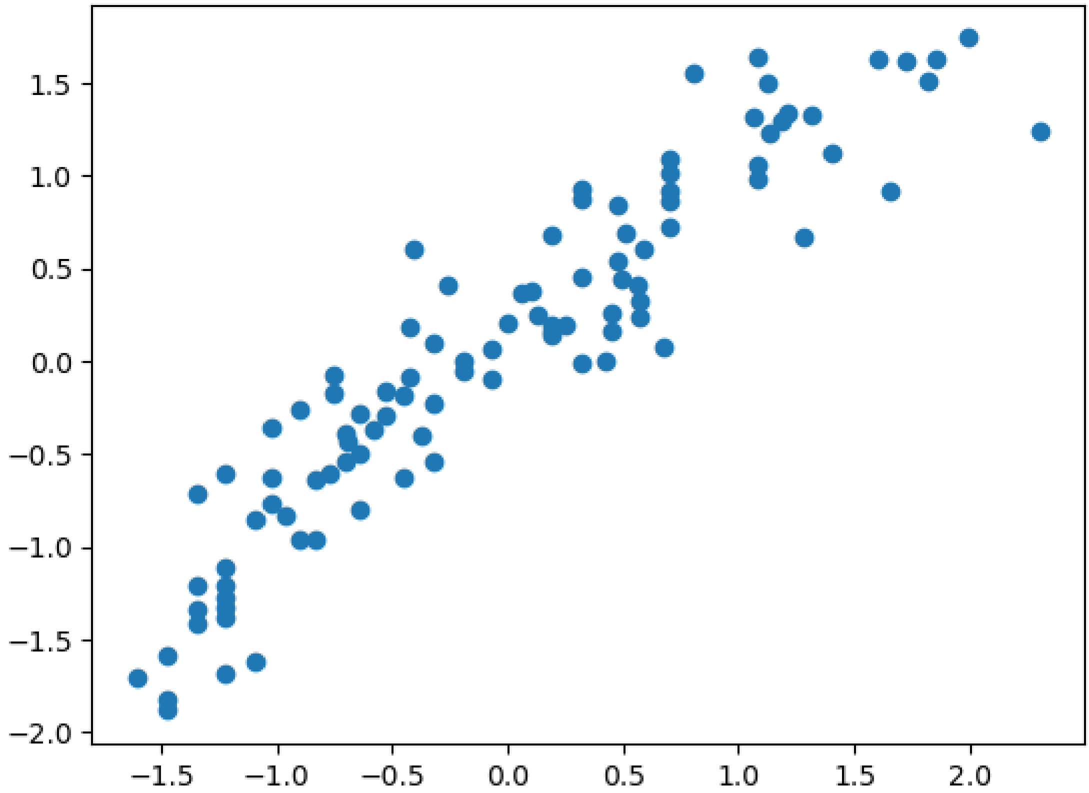


The x-axis represents the year of the car(independent variable). The y-axis represents the residuals, which are the differences between the actual car mpg and the predicted mpg. Each point on the scatterplot indicates how well the model's predictions match the actual mpg. If the points are randomly scattered around the horizontal line at y=0, it suggests that the model's predictions are unbiased and accurate. Horizontal Line at y=0:

The black line at y=0 indicates the reference line, representing zero residual. It helps to visually assess whether the residuals are centered around zero. If the points are evenly distributed above and below the line, it indicates that the model is making unbiased predictions on average.

In [68]:
```
plt.scatter(y_test['mpg'], Y_pred)
```

Out[68]: <matplotlib.collections.PathCollection at 0x7f8502f15bd0>



Insight:corelation between predicted and actual mpg is positive